FOUNDATION-0.6: Agent Registry - PART 1 (Code)

🎯 CONTEXT

Phase: FOUNDATION (Week 1 - Day 3 Morning)

Component: Orchestrator Agent Registry (Go)

Estimated Time: 15 min AI execution + 10 min verification

Complexity: MEDIUM

Risk Level: LOW

Files: Part 1 of 2 (Code implementation)

MILESTONE: Agent registration and discovery system! 🤝

📦 DEPENDENCIES

Must Complete First:

- **P-02:** Orchestrator Skeleton (Go) ✅ COMPLETED
- **FOUNDATION-0.2a-0.2e:** PostgreSQL schemas ✅ COMPLETED
- **FOUNDATION-0.3:** ClickHouse ✅ COMPLETED
- **FOUNDATION-0.4:** Qdrant ✅ COMPLETED

Required Services Running:

```bash
# Verify all services are healthy
cd ~/optiinfra
make verify

# Expected output:
# PostgreSQL... ✅ HEALTHY
# ClickHouse... ✅ HEALTHY
# Qdrant... ✅ HEALTHY
# Redis... ✅ HEALTHY
```

🎯 OBJECTIVE

Build **Agent Registry** system that enables:

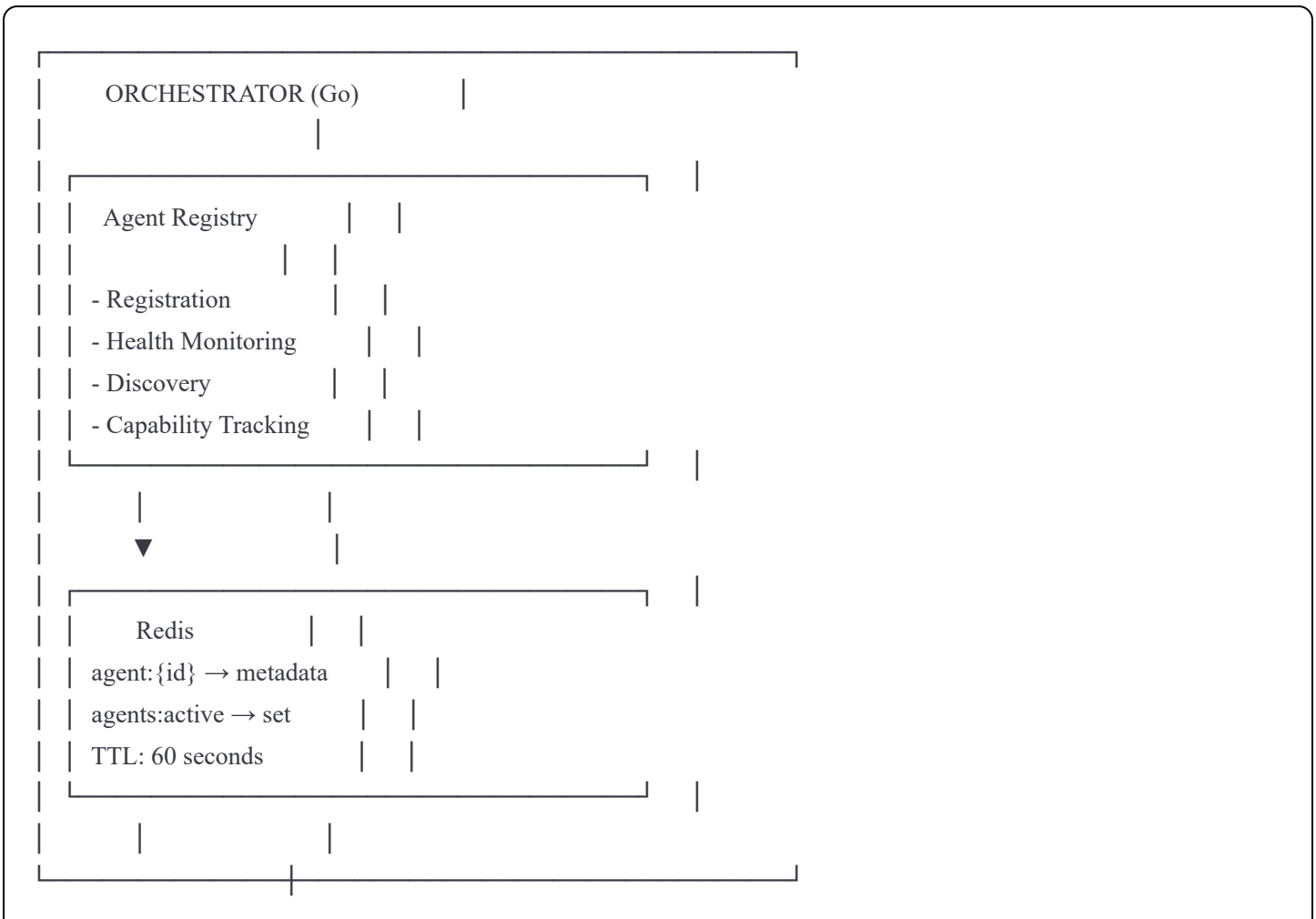
- ✅ Agents register themselves on startup
- ✅ Health monitoring and heartbeats
- ✅ Capability discovery
- ✅ Agent status tracking
- ✅ Automatic cleanup of dead agents

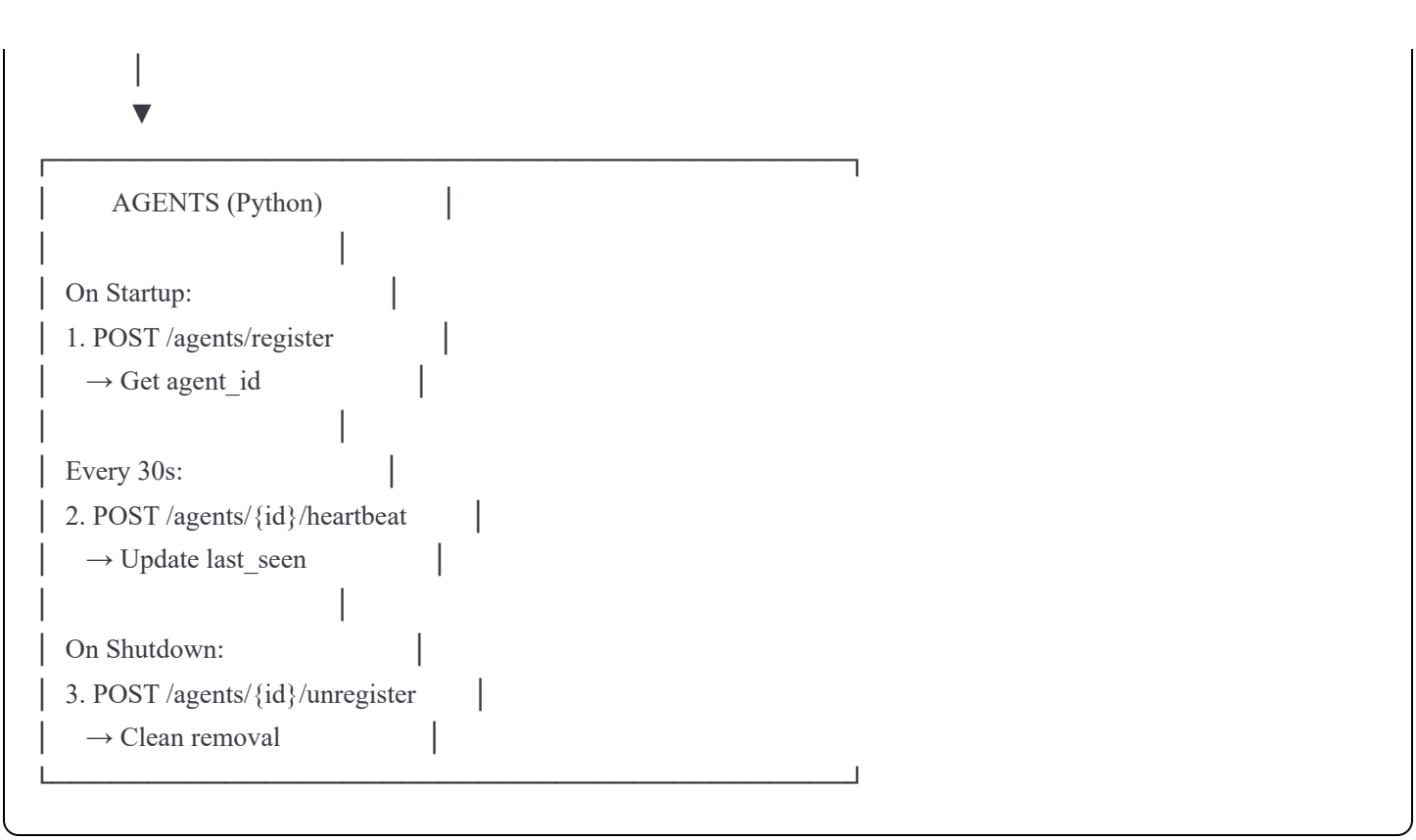
What We're Building:

Agent Registry Components:

1. **Registration API** - POST /agents/register
2. **Heartbeat API** - POST /agents/{id}/heartbeat
3. **Discovery API** - GET /agents, GET /agents/{id}
4. **Health Monitor** - Background goroutine checking agent health
5. **Redis Storage** - Fast agent lookup with TTL

Architecture:





Use Cases:

Agent Startup:

```
go

// Agent sends:
POST /agents/register
{
  "name": "cost-agent-1",
  "type": "cost",
  "capabilities": ["spot_migration", "reserved_instances"],
  "host": "cost-agent:8001",
  "status": "healthy"
}

// Orchestrator responds:
{
  "agent_id": "550e8400-e29b-41d4-a716-446655440000",
  "registered_at": "2025-10-20T10:30:00Z"
}
```

Request Routing (Future 0.7):

```
go
```

```
// Orchestrator needs a cost agent
agents := registry.GetAgentsByType("cost")
// Returns: [cost-agent-1, cost-agent-2]

// Pick healthy agent with capability
agent := registry.GetAgentWithCapability("cost", "spot_migration")
// Returns: cost-agent-1
```

📁 FILE 1: Agent Registry Models

Location: ~/optiinfra/services/orchestrator/internal/registry/models.go

```
go
```

```go
package registry

import (
  "time"
)

// AgentType represents the type of agent
type AgentType string

const (
  AgentTypeCost        AgentType = "cost"
  AgentTypePerformance AgentType = "performance"
  AgentTypeResource    AgentType = "resource"
  AgentTypeApplication AgentType = "application"
)

// AgentStatus represents the current status of an agent
type AgentStatus string

const (
  AgentStatusHealthy     AgentStatus = "healthy"
  AgentStatusDegraded    AgentStatus = "degraded"
  AgentStatusUnhealthy   AgentStatus = "unhealthy"
  AgentStatusUnreachable AgentStatus = "unreachable"
)

// Agent represents a registered agent
type Agent struct {
  ID           string        `json:"id"`
  Name         string        `json:"name"`
  Type         AgentType     `json:"type"`
  Host         string        `json:"host"`
  Port         int           `json:"port"`
  Capabilities []string      `json:"capabilities"`
  Status       AgentStatus   `json:"status"`
  Version      string        `json:"version"`
  RegisteredAt time.Time     `json:"registered_at"`
  LastSeen     time.Time     `json:"last_seen"`
  Metadata     map[string]interface{} `json:"metadata,omitempty"`
}

// RegistrationRequest is sent by agents to register
type RegistrationRequest struct {
```

```
    Name         string                `json:"name" binding:"required"`
    Type         AgentType             `json:"type" binding:"required"`
    Host         string                `json:"host" binding:"required"`
    Port         int                   `json:"port" binding:"required"`
    Capabilities []string              `json:"capabilities"`
    Version      string                `json:"version"`
    Metadata     map[string]interface{} `json:"metadata,omitempty"`
}

// RegistrationResponse is returned after successful registration
type RegistrationResponse struct {
    AgentID      string    `json:"agent_id"`
    RegisteredAt time.Time `json:"registered_at"`
    HeartbeatURL string    `json:"heartbeat_url"`
    Interval     int       `json:"heartbeat_interval_seconds"`
}

// HeartbeatRequest is sent by agents periodically
type HeartbeatRequest struct {
    Status   AgentStatus           `json:"status"`
    Metadata map[string]interface{} `json:"metadata,omitempty"`
}

// HeartbeatResponse confirms heartbeat received
type HeartbeatResponse struct {
    Received     bool      `json:"received"`
    NextInterval int       `json:"next_interval_seconds"`
    Timestamp    time.Time `json:"timestamp"`
}

// AgentListResponse returns list of agents
type AgentListResponse struct {
    Agents []Agent `json:"agents"`
    Count  int     `json:"count"`
}
```

📁 FILE 2: Agent Registry Core Logic

Location: `~/optiinfra/services/orchestrator/internal/registry/registry.go`

```
go
```

```go
package registry

import (
  "context"
  "encoding/json"
  "fmt"
  "log"
  "sync"
  "time"

  "github.com/go-redis/redis/v8"
  "github.com/google/uuid"
)

const (
  // Redis keys
  agentKeyPrefix     = "agent:"
  activeAgentsSetKey = "agents:active"

  // TTL for agent entries in Redis
  agentTTL = 60 * time.Second

  // Health check interval
  healthCheckInterval = 30 * time.Second

  // Heartbeat timeout (if no heartbeat for this long, mark unhealthy)
  heartbeatTimeout = 45 * time.Second
)

// Registry manages agent registration and discovery
type Registry struct {
  redis  *redis.Client
  ctx    context.Context
  mu     sync.RWMutex
  stopCh chan struct{}
}

// NewRegistry creates a new agent registry
func NewRegistry(redisClient *redis.Client) *Registry {
  return &Registry{
    redis:  redisClient,
    ctx:    context.Background(),
    stopCh: make(chan struct{}),
```

```
  }
}

// Start begins the health monitoring goroutine
func (r *Registry) Start() {
  go r.healthMonitor()
  log.Println("Agent registry started")
}

// Stop stops the health monitoring
func (r *Registry) Stop() {
  close(r.stopCh)
  log.Println("Agent registry stopped")
}

// Register registers a new agent
func (r *Registry) Register(req *RegistrationRequest) (*RegistrationResponse, error) {
  r.mu.Lock()
  defer r.mu.Unlock()

  // Generate agent ID
  agentID := uuid.New().String()

  // Create agent
  agent := &Agent{
    ID:           agentID,
    Name:         req.Name,
    Type:         req.Type,
    Host:         req.Host,
    Port:         req.Port,
    Capabilities: req.Capabilities,
    Status:       AgentStatusHealthy,
    Version:      req.Version,
    RegisteredAt: time.Now(),
    LastSeen:     time.Now(),
    Metadata:     req.Metadata,
  }

  // Store in Redis
  if err := r.storeAgent(agent); err != nil {
    return nil, fmt.Errorf("failed to store agent: %w", err)
  }

  // Add to active agents set
```

```
  if err := r.redis.SAdd(r.ctx, activeAgentsSetKey, agentID).Err(); err != nil {
    return nil, fmt.Errorf("failed to add to active set: %w", err)
  }

  log.Printf("Agent registered: %s (%s) - %s", agent.Name, agent.Type, agent.ID)

  return &RegistrationResponse{
    AgentID:      agentID,
    RegisteredAt: agent.RegisteredAt,
    HeartbeatURL: fmt.Sprintf("/agents/%s/heartbeat", agentID),
    Interval:     30, // heartbeat every 30 seconds
  }, nil
}

// Heartbeat updates agent's last seen time and status
func (r *Registry) Heartbeat(agentID string, req *HeartbeatRequest) (*HeartbeatResponse, error) {
  r.mu.Lock()
  defer r.mu.Unlock()

  // Get existing agent
  agent, err := r.getAgent(agentID)
  if err != nil {
    return nil, fmt.Errorf("agent not found: %w", err)
  }

  // Update status and last seen
  agent.LastSeen = time.Now()
  agent.Status = req.Status

  // Merge metadata
  if req.Metadata != nil {
    if agent.Metadata == nil {
      agent.Metadata = make(map[string]interface{})
    }
    for k, v := range req.Metadata {
      agent.Metadata[k] = v
    }
  }

  // Store updated agent
  if err := r.storeAgent(agent); err != nil {
    return nil, fmt.Errorf("failed to update agent: %w", err)
  }
```

```
  return &HeartbeatResponse{
    Received:     true,
    NextInterval: 30,
    Timestamp:    time.Now(),
  }, nil
}

// Unregister removes an agent from the registry
func (r *Registry) Unregister(agentID string) error {
  r.mu.Lock()
  defer r.mu.Unlock()

  // Remove from active set
  if err := r.redis.SRem(r.ctx, activeAgentsSetKey, agentID).Err(); err != nil {
    return fmt.Errorf("failed to remove from active set: %w", err)
  }

  // Delete agent key
  if err := r.redis.Del(r.ctx, agentKey(agentID)).Err(); err != nil {
    return fmt.Errorf("failed to delete agent: %w", err)
  }

  log.Printf("Agent unregistered: %s", agentID)
  return nil
}

// GetAgent retrieves a specific agent by ID
func (r *Registry) GetAgent(agentID string) (*Agent, error) {
  r.mu.RLock()
  defer r.mu.RUnlock()

  return r.getAgent(agentID)
}

// GetAllAgents retrieves all registered agents
func (r *Registry) GetAllAgents() ([]*Agent, error) {
  r.mu.RLock()
  defer r.mu.RUnlock()

  // Get all active agent IDs
  agentIDs, err := r.redis.SMembers(r.ctx, activeAgentsSetKey).Result()
  if err != nil {
    return nil, fmt.Errorf("failed to get active agents: %w", err)
  }
```

```
  agents := make([]*Agent, 0, len(agentIDs))
  for _, id := range agentIDs {
    agent, err := r.getAgent(id)
    if err != nil {
      log.Printf("Warning: failed to get agent %s: %v", id, err)
      continue
    }
    agents = append(agents, agent)
  }

  return agents, nil
}

// GetAgentsByType retrieves all agents of a specific type
func (r *Registry) GetAgentsByType(agentType AgentType) ([]*Agent, error) {
  allAgents, err := r.GetAllAgents()
  if err != nil {
    return nil, err
  }

  filtered := make([]*Agent, 0)
  for _, agent := range allAgents {
    if agent.Type == agentType {
      filtered = append(filtered, agent)
    }
  }

  return filtered, nil
}

// GetAgentWithCapability finds an agent with specific capability
func (r *Registry) GetAgentWithCapability(agentType AgentType, capability string) (*Agent, error) {
  agents, err := r.GetAgentsByType(agentType)
  if err != nil {
    return nil, err
  }

  // Filter by capability and status
  for _, agent := range agents {
    if agent.Status == AgentStatusHealthy {
      for _, cap := range agent.Capabilities {
        if cap == capability {
          return agent, nil
```

```
      }
    }
  }
 }

 return nil, fmt.Errorf("no healthy %s agent found with capability: %s", agentType, capability)
}

// GetHealthyAgents returns only healthy agents
func (r *Registry) GetHealthyAgents() ([]*Agent, error) {
 allAgents, err := r.GetAllAgents()
 if err != nil {
  return nil, err
 }

 healthy := make([]*Agent, 0)
 for _, agent := range allAgents {
  if agent.Status == AgentStatusHealthy {
   healthy = append(healthy, agent)
  }
 }

 return healthy, nil
}

// ============================================================================
// INTERNAL HELPERS
// ============================================================================

func (r *Registry) storeAgent(agent *Agent) error {
 data, err := json.Marshal(agent)
 if err != nil {
  return fmt.Errorf("failed to marshal agent: %w", err)
 }

 // Store with TTL
 if err := r.redis.Set(r.ctx, agentKey(agent.ID), data, agentTTL).Err(); err != nil {
  return fmt.Errorf("failed to store in redis: %w", err)
 }

 return nil
}

func (r *Registry) getAgent(agentID string) (*Agent, error) {
```

```
  data, err := r.redis.Get(r.ctx, agentKey(agentID)).Result()
  if err == redis.Nil {
    return nil, fmt.Errorf("agent not found")
  } else if err != nil {
    return nil, fmt.Errorf("failed to get from redis: %w", err)
  }

  var agent Agent
  if err := json.Unmarshal([]byte(data), &agent); err != nil {
    return nil, fmt.Errorf("failed to unmarshal agent: %w", err)
  }

  return &agent, nil
}

func agentKey(agentID string) string {
  return agentKeyPrefix + agentID
}

// ============================================================================
// HEALTH MONITORING
// ============================================================================

func (r *Registry) healthMonitor() {
  ticker := time.NewTicker(healthCheckInterval)
  defer ticker.Stop()

  for {
    select {
    case <-ticker.C:
      r.checkAgentHealth()
    case <-r.stopCh:
      return
    }
  }
}

func (r *Registry) checkAgentHealth() {
  r.mu.Lock()
  defer r.mu.Unlock()

  agents, err := r.GetAllAgents()
  if err != nil {
    log.Printf("Error checking agent health: %v", err)
```

```
    return
  }

  now := time.Now()
  for _, agent := range agents {
    timeSinceLastSeen := now.Sub(agent.LastSeen)

    // Mark unhealthy if no heartbeat for too long
    if timeSinceLastSeen > heartbeatTimeout {
      if agent.Status != AgentStatusUnreachable {
        log.Printf("Agent %s (%s) is unreachable - last seen %v ago",
          agent.Name, agent.ID, timeSinceLastSeen)
        agent.Status = AgentStatusUnreachable
        if err := r.storeAgent(agent); err != nil {
          log.Printf("Failed to update agent status: %v", err)
        }
      }
    }
  }
}
```

📁 FILE 3: HTTP Handlers

Location: ~/optiinfra/services/orchestrator/internal/registry/handlers.go

```
go
```

```go
package registry

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

// Handler provides HTTP handlers for the registry
type Handler struct {
  registry *Registry
}

// NewHandler creates a new handler
func NewHandler(registry *Registry) *Handler {
  return &Handler{
    registry: registry,
  }
}

// RegisterRoutes registers all registry routes
func (h *Handler) RegisterRoutes(router *gin.Engine) {
  agents := router.Group("/agents")
  {
    agents.POST("/register", h.Register)
    agents.POST("/:id/heartbeat", h.Heartbeat)
    agents.POST("/:id/unregister", h.Unregister)
    agents.GET("", h.List)
    agents.GET("/:id", h.Get)
    agents.GET("/type/:type", h.ListByType)
  }
}

// Register handles agent registration
func (h *Handler) Register(c *gin.Context) {
  var req RegistrationRequest
  if err := c.ShouldBindJSON(&req); err != nil {
    c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    return
  }

  resp, err := h.registry.Register(&req)
  if err != nil {
```

```
    c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
    return
  }

  c.JSON(http.StatusCreated, resp)
}

// Heartbeat handles agent heartbeat
func (h *Handler) Heartbeat(c *gin.Context) {
  agentID := c.Param("id")

  var req HeartbeatRequest
  if err := c.ShouldBindJSON(&req); err != nil {
    c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    return
  }

  resp, err := h.registry.Heartbeat(agentID, &req)
  if err != nil {
    c.JSON(http.StatusNotFound, gin.H{"error": err.Error()})
    return
  }

  c.JSON(http.StatusOK, resp)
}

// Unregister handles agent unregistration
func (h *Handler) Unregister(c *gin.Context) {
  agentID := c.Param("id")

  if err := h.registry.Unregister(agentID); err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
    return
  }

  c.JSON(http.StatusOK, gin.H{"message": "Agent unregistered successfully"})
}

// List returns all registered agents
func (h *Handler) List(c *gin.Context) {
  agents, err := h.registry.GetAllAgents()
  if err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
    return
```

```
  }

  c.JSON(http.StatusOK, AgentListResponse{
    Agents: convertToAgentSlice(agents),
    Count:  len(agents),
  })
}

// Get returns a specific agent
func (h *Handler) Get(c *gin.Context) {
  agentID := c.Param("id")

  agent, err := h.registry.GetAgent(agentID)
  if err != nil {
    c.JSON(http.StatusNotFound, gin.H{"error": "Agent not found"})
    return
  }

  c.JSON(http.StatusOK, agent)
}

// ListByType returns agents of a specific type
func (h *Handler) ListByType(c *gin.Context) {
  agentType := AgentType(c.Param("type"))

  agents, err := h.registry.GetAgentsByType(agentType)
  if err != nil {
    c.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
    return
  }

  c.JSON(http.StatusOK, AgentListResponse{
    Agents: convertToAgentSlice(agents),
    Count:  len(agents),
  })
}

func convertToAgentSlice(agents []*Agent) []Agent {
  result := make([]Agent, len(agents))
  for i, agent := range agents {
    result[i] = *agent
  }
```

```
    return result
}
```

📁 FILE 4: Update Main Orchestrator

Location: `~/optiinfra/services/orchestrator/cmd/server/main.go`

```go
```

```go
package main

import (
  "context"
  "log"
  "os"
  "os/signal"
  "syscall"
  "time"

  "github.com/gin-gonic/gin"
  "github.com/go-redis/redis/v8"

  "optiinfra/services/orchestrator/internal/registry"
)

func main() {
  // Initialize Redis
  redisClient := redis.NewClient(&redis.Options{
    Addr:     getEnv("REDIS_ADDR", "localhost:6379"),
    Password: getEnv("REDIS_PASSWORD", ""),
    DB:       0,
  })

  // Test Redis connection
  ctx := context.Background()
  if err := redisClient.Ping(ctx).Err(); err != nil {
    log.Fatal("Failed to connect to Redis:", err)
  }
  log.Println("Connected to Redis")

  // Initialize Agent Registry
  agentRegistry := registry.NewRegistry(redisClient)
  agentRegistry.Start()
  defer agentRegistry.Stop()

  // Initialize Gin
  router := gin.Default()

  // Health check endpoint
  router.GET("/health", func(c *gin.Context) {
    c.JSON(200, gin.H{
      "status": "healthy",
```

```
    "service": "orchestrator",
    "timestamp": time.Now(),
  })
 })

 // Register agent registry routes
 registryHandler := registry.NewHandler(agentRegistry)
 registryHandler.RegisterRoutes(router)

 // Start server
 port := getEnv("PORT", "8080")
 log.Printf("Starting orchestrator on port %s", port)

 // Graceful shutdown
 srv := &http.Server{
  Addr:    ":" + port,
  Handler: router,
 }

 go func() {
  if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
   log.Fatalf("Server failed: %v", err)
  }
 }()

 // Wait for interrupt signal
 quit := make(chan os.Signal, 1)
 signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
 <-quit

 log.Println("Shutting down server...")

 // Graceful shutdown with timeout
 ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
 defer cancel()

 if err := srv.Shutdown(ctx); err != nil {
  log.Fatal("Server forced to shutdown:", err)
 }

 log.Println("Server exited")
}

func getEnv(key, defaultValue string) string {
```

```
  if value := os.Getenv(key); value != "" {
    return value
  }
  return defaultValue
}
```

📁 FILE 5: Go Module Dependencies

Location: ~/optiinfra/services/orchestrator/go.mod

```
go

module optiinfra/services/orchestrator

go 1.21

require (
  github.com/gin-gonic/gin v1.9.1
  github.com/go-redis/redis/v8 v8.11.5
  github.com/google/uuid v1.5.0
)
```

📁 FILE 6: Python Agent Registration Helper

Location: ~/optiinfra/shared/orchestrator/registration.py

```
python
```

```
"""
Agent registration helper for Python agents.

Usage:
    from shared.orchestrator.registration import AgentRegistration

    registration = AgentRegistration(
        agent_name="cost-agent-1",
        agent_type="cost",
        host="localhost",
        port=8001,
        capabilities=["spot_migration", "reserved_instances"]
    )

    # Register on startup
    registration.register()

    # Start heartbeat loop
    registration.start_heartbeat()

    # Unregister on shutdown
    registration.unregister()
"""

import requests
import threading
import time
import logging
from typing import List, Dict, Any, Optional

logger = logging.getLogger(__name__)


class AgentRegistration:
    """Handles agent registration with the orchestrator."""

    def __init__(
        self,
        agent_name: str,
        agent_type: str,
        host: str,
        port: int,
        capabilities: List[str],
```

```
    orchestrator_url: str = "http://localhost:8080",
    version: str = "1.0.0",
    metadata: Optional[Dict[str, Any]] = None
):
    self.agent_name = agent_name
    self.agent_type = agent_type
    self.host = host
    self.port = port
    self.capabilities = capabilities
    self.orchestrator_url = orchestrator_url
    self.version = version
    self.metadata = metadata or {}

    self.agent_id: Optional[str] = None
    self.heartbeat_interval: int = 30
    self.heartbeat_thread: Optional[threading.Thread] = None
    self.stop_heartbeat = threading.Event()

def register(self) -> bool:
    """
    Register this agent with the orchestrator.

    Returns:
        bool: True if registration successful
    """
    try:
        response = requests.post(
            f"{self.orchestrator_url}/agents/register",
            json={
                "name": self.agent_name,
                "type": self.agent_type,
                "host": self.host,
                "port": self.port,
                "capabilities": self.capabilities,
                "version": self.version,
                "metadata": self.metadata
            },
            timeout=10
        )

        if response.status_code == 201:
            data = response.json()
            self.agent_id = data["agent_id"]
            self.heartbeat_interval = data.get("heartbeat_interval_seconds", 30)
```

```
            logger.info(f"Agent registered successfully: {self.agent_id}")
            return True
        else:
            logger.error(f"Registration failed: {response.status_code} - {response.text}")
            return False

    except Exception as e:
        logger.error(f"Registration error: {e}")
        return False

def start_heartbeat(self):
    """Start the heartbeat thread."""
    if not self.agent_id:
        logger.error("Cannot start heartbeat - agent not registered")
        return

    self.stop_heartbeat.clear()
    self.heartbeat_thread = threading.Thread(target=self._heartbeat_loop, daemon=True)
    self.heartbeat_thread.start()
    logger.info("Heartbeat started")

def stop_heartbeat_loop(self):
    """Stop the heartbeat thread."""
    self.stop_heartbeat.set()
    if self.heartbeat_thread:
        self.heartbeat_thread.join(timeout=5)
    logger.info("Heartbeat stopped")

def unregister(self):
    """Unregister this agent from the orchestrator."""
    if not self.agent_id:
        return

    try:
        # Stop heartbeat first
        self.stop_heartbeat_loop()

        # Unregister
        response = requests.post(
            f"{self.orchestrator_url}/agents/{self.agent_id}/unregister",
            timeout=10
        )

        if response.status_code == 200:
```

```
            logger.info(f"Agent unregistered successfully: {self.agent_id}")
        else:
            logger.warning(f"Unregister failed: {response.status_code}")

    except Exception as e:
        logger.error(f"Unregister error: {e}")

def _heartbeat_loop(self):
    """Background thread that sends periodic heartbeats."""
    while not self.stop_heartbeat.is_set():
        try:
            self._send_heartbeat()
        except Exception as e:
            logger.error(f"Heartbeat error: {e}")

        # Wait for next interval
        self.stop_heartbeat.wait(self.heartbeat_interval)

def _send_heartbeat(self):
    """Send a single heartbeat to the orchestrator."""
    if not self.agent_id:
        return

    try:
        response = requests.post(
            f"{self.orchestrator_url}/agents/{self.agent_id}/heartbeat",
            json={
                "status": "healthy",
                "metadata": self.metadata
            },
            timeout=5
        )
```